2018 IEEE International Symposium on High Performance Computer Architecture

OuterSPACE: An Outer Product based Sparse Matrix Multiplication Accelerator

Subhankar Pal* Jonathan Beaumont* Dong-Hyeon Park* Aporva Amarnath* Siying Feng*
Chaitali Chakrabarti† Hun-Seok Kim* David Blaauw* Trevor Mudge* Ronald Dreslinski*

*University of Michigan, Ann Arbor, MI †Arizona State University, Tempe, AZ
*{subh, jbbeau, dohypark, aporvaa, fengsy, hunseok, blaauw, tnm, rdreslin}@umich.edu
†chaitali@asu.edu

ABSTRACT

Sparse matrices are widely used in graph and data analytics, machine learning, engineering and scientific applications. This paper describes and analyzes OuterSPACE, an accelerator targeted at applications that involve large sparse matrices. OuterSPACE is a highly-scalable, energy-efficient, reconfigurable design, consisting of massively parallel Single Program, Multiple Data (SPMD)-style processing units, distributed memories, high-speed crossbars and High Bandwidth Memory (HBM).

We identify redundant memory accesses to non-zeros as a key bottleneck in traditional sparse matrix-matrix multiplication algorithms. To ameliorate this, we implement an outer product based matrix multiplication technique that eliminates redundant accesses by decoupling multiplication from accumulation. We demonstrate that traditional architectures, due to limitations in their memory hierarchies and ability to harness parallelism in the algorithm, are unable to take advantage of this reduction without incurring significant overheads. OuterSPACE is designed to specifically overcome these challenges.

We simulate the key components of our architecture using gem5 on a diverse set of matrices from the University of Florida's SuiteSparse collection and the Stanford Network Analysis Project and show a mean speedup of 7.9× over Intel Math Kernel Library on a Xeon CPU, 13.0× against cuSPARSE and 14.0× against CUSP when run on an NVIDIA K40 GPU, while achieving an average throughput of 2.9 GFLOPS within a 24 W power budget in an area of 87 mm^2.

KEYWORDS

Sparse matrix processing, application-specific hardware, parallel computer architecture, hardware-software co-design, hardware accelerators

1. INTRODUCTION

Generalized sparse matrix-matrix multiplication (SpGEMM) and sparse matrix-vector multiplication (SpMV) are two key kernels of complex operations in domains such as graph analytics, machine learning, and scientific computation, as we elaborate in Section 2. The percentage of non-zero elements in the matrices involved can be very small. For example, the number of active daily Facebook users is currently 1.08 billion and the average number of friends per user is 338 [56]. A graph representing Facebook users as vertices and "friendships" between users as edges results in an adjacency matrix of dimension 1.08 billion with a density of just 0.0003%.

Sparse matrix-based computations are becoming an increasingly important problem. These applications are typically bottlenecked by memory rather than computation, primarily due to irregular, data-dependent memory accesses in existing matrix libraries, leading to poor throughput and performance [25, 43].

The demise of Moore's Law has led to renewed interest in accelerators to improve performance and reduce energy and cost. In order to address these issues for applications involving sparse matrix computations, we propose a custom accelerator, OuterSPACE, that consists of asynchronous Single Program, Multiple Data (SPMD)-style processing units with dynamically-reconfigurable non-coherent caches and crossbars. OuterSPACE is designed to work with the unconventional outer product based matrix multiplication approach [16, 60], which involves multiplying the i^{th} column of the first matrix ($\mathbf{A}$) with the i^{th} row of the second matrix ($\mathbf{B}$), for all i. Each multiplication generates a partial product matrix and all the generated matrices are then accumulated element-wise to form the final result. A seemingly obvious drawback of this approach is the maintenance and storage of these partial product matrices. In the case of sparse matrices, however, this is much less of a concern and other considerations dominate. We identify contrasting data-sharing patterns in the two distinct, highly parallel compute phases: *multiply* and *merge*. The *multiply* phase involves data-sharing across parallel computation streams, while the *merge* phase involves strictly independent processing with little-to-no communication or synchronization between streams. This discrepancy leads to sub-optimal execution of the outer product method on mainstream architectures, namely GPUs and multi-core/many-core CPUs (Section 4.4).

The reconfigurability of OuterSPACE enables us to meet the contrasting computational needs of the outer product method's two compute phases. Employing asynchronous *SPMD-style* processing elements allows for

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00067

control-divergent code to operate fully in parallel, as opposed to SIMD architectures, which need to at least partially serialize it. Software-controlled *scratchpads*, coupled with hardware-controlled *caches*, prevent wasted data accesses to the main memory. Further, allowing *non-coherence* relieves pressure on the memory system associated with excess broadcasts and writebacks (which can contribute up to 30-70% of the total write traffic [38]), providing a fraction of the performance benefits.

While the main focus of our work is the acceleration of sparse matrix-matrix multiplication, we also present results of sparse matrix-vector multiplication and describe how element-wise operations can be performed on the OuterSPACE system.

We use a server-class multi-core CPU and GPU as baselines to compare our architecture against. For sparse matrix multiplication on the CPU, we use state-of-the-art sparse BLAS functions from Intel Math Kernel Library (MKL). MKL provides math routines for applications that solve large computational problems and are extensively parallelized by OpenMP threading while using vector operations provided by the AVX instruction set. For the GPU, we compare our architecture against the cuSPARSE and CUSP libraries. cuSPARSE [2] applies row-by-row parallelism and uses a hash table to merge partial products for each row of the output matrix. CUSP [10, 19] presents fined-grained parallelism by accessing the input matrices row-by-row and storing the partial result with possible duplicates into an intermediate coordinate format. The intermediate structure is then sorted and compressed into the output matrix.

The rest of the paper is organized as follows. *Section 2* discusses a wide spectrum of applications that utilize sparse matrix operation kernels. *Section 3* provides a brief background on inner and outer product multiplication and sparse matrix storage formats. *Section 4* discusses our outer product implementations for sparse matrix-matrix multiplication and evaluates their performance on the CPU and GPU. *Section 5* provides details of the OuterSPACE architecture and how the outer product algorithm efficiently maps to it. *Section 6* presents our experimental setup and *Section 7* presents results and insights drawn from them. *Section 8* briefly discusses how OuterSPACE can be scaled up to support larger matrices. Lastly, *Section 9* discusses related work and *Section 10* provides a few concluding remarks.

2. MOTIVATION

Sparse matrices are ubiquitous in most modern applications that operate on big data. It is the general consensus that a selection of linear algebra routines optimized over years of research can be used to accelerate a wide range of graph and scientific algorithms [22].

Sparse matrix-matrix multiplication, in particular, is a significant building block of multiple algorithms prevalent in graph analytics, such as breadth-first search [23][24], matching [49], graph contraction [15], peer pressure clustering [54], cycle detection [65], Markov clustering [61], and triangle counting [9]. It is also a key kernel in many scientific-computing applications. For example, fast sparse matrix-matrix multiplication is a performance bottleneck in the hybrid linear solver applying the Schur complement method [63] and algebraic multigrid methods [10]. Other computing applications, such as color intersection searching [33], context-free grammar parsing [48], finite element simulations based on domain decomposition [27], molecular dynamics [30], and interior point methods [34] also rely heavily on sparse matrix-matrix multiplication.

Sparse matrix-vector multiplication is also predominant across diverse applications, such as PageRank [14], minimal spanning tree, single-source shortest path and vertex/edge-betweenness centrality calculations. It serves as a dominant compute primitive in Machine Learning (ML) algorithms such as support vector machine [47] and ML-based text analytics applications [44].

While GPUs demonstrate satisfactory compute efficiency on sparse matrix-vector multiplication and sufficiently dense matrix-matrix multiplication [11], we show that compute units are significantly underutilized when the density drops below 0.1%, often achieving fewer than 1 GFLOPS, despite a peak theoretical throughput of over 4 TFLOPS [59]. This is further supported by the fact that rankings, such as the Green Graph 500 list [4], are dominated by CPU-based systems.

For large dense matrices, inner product multiplication, block partitioning and tiling techniques are used to take advantage of data locality. However, when the density of the input matrices is decreased, the run-time is dominated by irregular memory accesses and index-matching in order to perform the inner product operations. Moreover, while tiling techniques can reduce redundant reads to main memory in the short-term, the on-chip storage constraints still necessitate that many data elements be redundantly fetched multiple times across tiles [31]. The stark inefficiencies on both the hardware and algorithmic fronts motivate our work to formulate a new approach for sparse matrix multiplication acceleration.

3. BACKGROUND

This section outlines a few fundamental concepts behind matrix-matrix multiplication and storage formats for representation of sparse matrices in memory.

3.1 Matrix Multiplication

3.1.1 The Inner Product Approach

Traditionally, generalized matrix-matrix multiplication (GEMM) is performed using the inner product approach. This is computed using a series of dot product operations between rows of the first matrix ($\mathbf{A}$) and columns of the second matrix ($\mathbf{B}$) and results in elements of the final product ($\mathbf{C}$):

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \times b_{k,j}$$

Here, N is the number of columns in $\mathbf{A}$ (or rows in $\mathbf{B}$), while i and j are the row and column indices, respectively, of an element in the final matrix. Thus, each element of the final matrix is computed through a series of multiply-and-accumulate (MAC) operations.

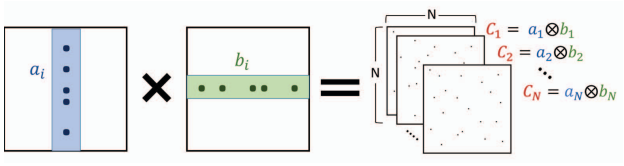

Figure 1: Outer product multiplication of matrices **A** and **B**. Each column-of-**A** and the corresponding row-of-**B** are multiplied with each other to produce N partial product matrices, $\mathbf{C_i}$. These are then summed together to produce the final result matrix **C**.

This is generally optimized using block partitioning and tiling techniques [62].

3.1.2 *The Outer Product Approach*

The outer product method [16, 60] multiples two matrices **A** and **B** by decomposing the operation into *outer product multiplications* of pairs of columns-of-**A** and rows-of-**B**, as illustrated in Figure 1. Mathematically,

$$\mathbf{C} = \sum_{i=0}^{N-1} \mathbf{C}_i = \sum_{i=0}^{N-1} \mathbf{a}_i \mathbf{b}_i$$

where $\mathbf{a}_i$ is the i^{th} column-of-**A**, $\mathbf{b}_i$ is the i^{th} row-of-**B** and $\mathbf{C}_i$ is a partial product matrix. Thus, the computation is divided into two sets: *multiply* operations to generate the partial products, followed by *merge* operations to accumulate the partial products into the final result. In Section 4, we propose an outer product based sparse matrix multiplication paradigm based on this.

3.2 Compressed Storage Formats

An **M**×**N** matrix is often represented in the dense format as a 2-D array laid out in the memory as an **M**×**N** contiguous block. For sparse matrices, however, most of the elements are zeros, and hence, there is little merit in storing such matrices in the dense format.

The Compressed Sparse Row (CSR) format represents a matrix in a compressed manner using three arrays. The *vals* array consists of the non-zero elements of the matrix in row-major order, the *cols* array contains the column indices of the elements in *vals*, the *row-ptrs* array contains pointers to the start of each row of the matrix in the *cols* and *vals* arrays. The dual of the CSR format is the Compressed Sparse Column (CSC) format, which is comprised of the *vals*, *rows* and *col-ptrs* arrays.

In our implementation, while not being restrictive, we employ a similar storage scheme, consisting of a contiguous block of row pointers each pointing to contiguous arrays of column index-value pairs. We henceforth refer to this as the Compressed Row (CR) format. The complementary format, Compressed Column (CC) format, consists of column pointers pointing to arrays of row index-value pairs.

4. OUTER PRODUCT IMPLEMENTATION

This section details the highly parallel outer product algorithm alluded to in Section 3.1.2, which maximizes memory reuse and avoids redundant reads to non-zeros.

A major inefficiency of sparse inner product multiplication (i.e., row-of-**A** × column-of-**B**) is that multiplications are performed selectively on matched non-zero indices. The outer product technique benefits over other conventional methods of multiplying matrices through:

- *Elimination of index-matching*: Each pair of non-zero elements from column-of-**A** and the corresponding row-of-**B** produce meaningful outputs. This is in contrast to inner product like algorithms, where the indices need to be matched before multiplication, leading to inefficient utilization of memory bandwidth to fetch elements redundantly.
- *Maximized reuse of non-zeros*: All elements in a row-of-**B** are shared for all elements in a column-of-**A** within an outer product. This maximizes the amount of reuse within a particular outer product calculation to its theoretical maximum, as we illustrate later in Figure 2.
- *Minimized loads of a column and row*: As a result of maximized data reuse within a outer product calculation, we have no available data reuse across different outer products. Thus, once the computation between a column-of-**A** and the corresponding row-of-**B** is completed, they are never used again and can be evicted from local memory.

In the rest of the section, we present details about the two phases of outer product multiplication: *multiply* and *merge*. Since our algorithm requires that **A** be in CC format and **B** be in CR, we describe in Section 4.3 how we convert a matrix into its complementary format.

4.1 Multiply Phase

Figure 2 shows an example multiplication of two 4×4 sparse matrices, given three parallel processing units in the system. For clarity of understanding, the dense representations of matrices **A** and **B** are shown on the top-left of the figure. These matrices are decomposed into pairs of columns-of-**A** and rows-of-**B**. An outer

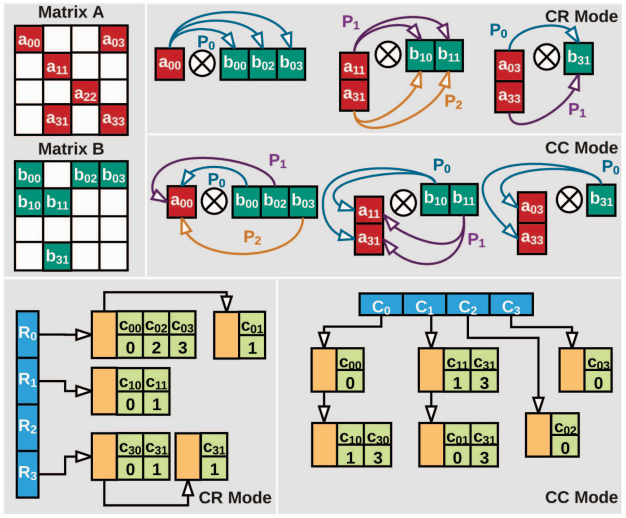

Figure 2: Outer product multiplication of matrices **A** (in CC) and **B** (in CR), illustrated in dense format, using three processing elements, and the layout of the partial products in memory. Both the CR and the CC modes of operation are shown here. Note that the third row of **B** is empty and hence no outer product is formed corresponding to it. The blue blocks represent the row/column pointers and the orange + green blocks are the partial product rows/columns containing index-value pairs.

product operation between each pair generates a full 4×4 compressed matrix and each of the generated matrices are summed together to produce the final result. In the CR mode of operation, each processing unit multiplies one non-zero element from a column-of-**A** with all the non-zeros in the corresponding row-of-**B**. The processing units are greedily scheduled in this example.

In our implementation, we store the intermediate partial products as a set of linked lists corresponding to each row (pointed to by R_i), where each node of the list contains a contiguous set of values representing a partial row of an outer product (Figure 2).

The CC mode of operation is analogous to the CR mode, where we re-program the processing units to multiply an element of a row-of-**B** with all the non-zeros in the corresponding column of **A**. The row pointers, R_i are replaced by column pointers, C_i. This is illustrated in the bottom-right part of Figure 2.

4.2 Merge Phase

The outer products pointed to by a row/column pointer need to be merged to form the final result. We assign the processing units to walk through the linked list pointed to by R_i/C_i and merge them to form a complete final row/column. In the event that multiple data values from different outer products correspond to the same index, they must be summed together. However, this gets increasingly rare with sparser matrices.

In Section 5, we elaborate the merging scheme that maps efficiently to the architecture of OuterSPACE. The hardware can be programmed to produce the resultant matrix in either the CR or the CC format. For brevity, we assume CR mode operation in the rest of the paper.

4.3 Matrix Format Conversion

When matrices **A** and **B** are not available in the CC and CR formats, respectively, either one or both will have to be converted to the complementary format. This is a one-time requirement for chained multiplication operations of the type $\mathbf{A}\times\mathbf{B}\times\mathbf{C}\ldots$, since OuterSPACE can output the result in either CR or CC formats. However, computations such as $\mathbf{A}^N$ can be decomposed into a logarithmic number of operations ($\mathbf{A}^2=\mathbf{A}\times\mathbf{A}$, $\mathbf{A}^4=\mathbf{A}^2\times\mathbf{A}^2$ and so on), where each operation would consist of conversion followed by actual computation. The requirement of conversion is obviated for *symmetric* matrices, since the CR and CC forms are equivalent.

In our evaluations, we assume that both the inputs, **A** and **B**, are available in the CR format, such that **A** must be converted. We partition the conversion operation into *conversion-load* and *conversion-merge* phases, analogous to the *multiply* and *merge* phases. The processing elements stream through **A** and store it into the intermediate data structure (Figure 2) in parallel. Conceptually, this is similar to multiplying Matrix **A** with an *Identity Matrix* of the same dimension:

$$\mathbf{I_{CC}}\times\mathbf{A_{CR}} \rightarrow \mathbf{A_{CC}} \text{ (CC mode)}$$

where, $\mathbf{I_{CC}}$ is the *Identity Matrix* and the subscripts represent the respective storage formats.

4.4 Performance on Traditional Hardware

Outer product multiplication is a well-established linear algebra routine. Yet, it is not widely implemented in mainstream hardware, as these designs are not well-suited for such algorithms. We expose the inefficiencies of this paradigm on traditional hardware by quantitatively comparing our outer product implementations against state-of-the art libraries, due to the absence of readily available libraries based on outer product multiplication.

4.4.1 Multi-Core CPU (Intel Xeon)

Figure 3 compares the execution times of our CPU implementation of the outer product algorithm, using the POSIX threads library, against the Intel MKL SpGEMM library, on an Intel Xeon processor with 6 threads.

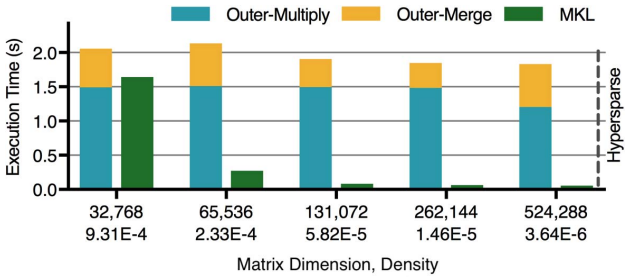

Figure 3: Comparison of our outer product implementation against Intel MKL on a Xeon multi-core CPU. The matrices are uniformly random with increasing dimension and decreasing density, keeping the number of non-zeros constant at 1 million. Format conversion and memory allocation times are not considered.

While the execution time of MKL drops exponentially with decreasing matrix density, the outer product algorithm must overcome two overheads with increasing matrix dimension (N): the decreasing number of useful operations performed at each matrix datum and the increasing number of book-keeping operations due to the growing size of the data structure in Figure 2. Thus, the price of no index-matching and minimized redundant reads of non-zeros in the outer product technique is paid for by additional pressure on the memory system, as N $N\times N$ partial product matrices are streamed out during the *multiply* phase and back in during the *merge* phase.

This necessitates larger memory bandwidth and more cores to churn through the data streams than available on our 6-core CPU. It is further exacerbated by the absence of software-controlled scratchpad memories and the ineffectiveness of CPU caching for the *merge* phase, which does not exhibit any data sharing within caches and thus leads to thrashing. This is substantiated by our studies of cache performance of the matrices in Figure 3, which show mean L2 hit rates of 0.14 and 0.12 during the *multiply* and *merge* phases, respectively. In comparison, MKL spGEMM routines are vectorized and heavily optimized for the multi-core architecture.

Table 1 presents data generated using Intel VTune Amplifier for a Core i7 CPU running the MKL on the same matrices as in Figure 3. The under-utilization of bandwidth (average of 62%) suggests that bandwidth is not the primary bottleneck for the MKL and increasing it will likely provide only sub-linear speedups.

Table 1: Bandwidth utilization of the MKL sparse GEMM on an Intel Core i7 running 4 threads. Each matrix has a uniform random distribution of 10 million non-zeros.

Matrix Dimension	Peak Bandwidth Utilization (%)	Avg. Bandwidth Utilization (%)
1,048,576	62.5	44.2
2,097,152	67.5	58.4
4,194,304	67.5	62.0
8,388,608	85.0	62.4

4.4.2 GPU (NVIDIA Tesla)

NVIDIA's implementations for matrix multiplication provided in their CUSP and cuSPARSE libraries perform very poorly at density levels below 0.01%. Running synthetic workloads at this level of sparsity achieves fewer than 1 GFLOPS. L1 cache hit rate and utilized memory bandwidth drop to under 40% and 10 GB/s, from 86% and 30 GB/s at 10% density. We compare the performance of our custom outer product implementation, written in CUDA, against CUSP. Our implementation makes use of the GPU's available scratchpad storage. Figure 4 compares the execution times when run on an NVIDIA K40 GPU.

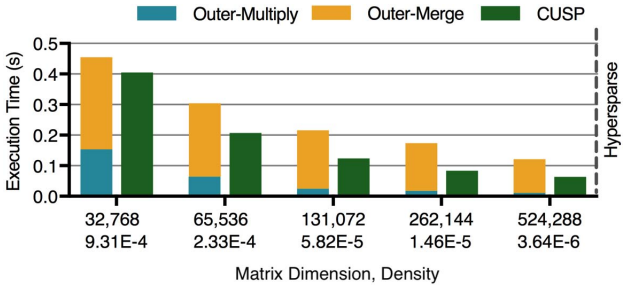

Figure 4: Comparison of a GPU outer product implementation against CUSP. The matrices are uniform random with increasing size while density is decreased, keeping the number of non-zeros constant at 1 million.

A comparison of results from Figure 3 and Figure 4 show that the GPU makes better use of available processing power and bandwidth than the CPU. The multiplication phase streams and processes the data much faster than the CPU implementation, scaling roughly linearly with decreasing density.

However, latency is quickly dominated by the merge phase. Despite both phases achieving similarly high L1 hit rates ($> 80\%$) and low data dependency stalls ($< 5\%$), the merge phase suffers from a much lower total throughput. This is a result of numerous conditional branches within the code to handle different relative column indices as they are read in and sorted. Because there is little correlation between adjacent threads as they process these branches, many threads within a given warp diverge and must be executed serially. Thus, while the high degree of parallelism available is attractive, the SIMD nature of the GPU's processing elements prevent an overall win of the algorithm over traditional libraries.

4.4.3 Many-Core CPU (Intel Xeon Phi)

Our experiments with the CPU outer product code on an Intel Xeon Phi Knights Corner system show an average slowdown of 14.7× compared to the CPU, for uniformly random matrices of dimensions varying from 32K to 524K with the number of non-zeros fixed at 1 million. We also note that denser matrices incur a significant amount of memory allocation overhead, which worsens the overall execution time. Although the outer product approach has high degrees of parallelism, it lacks an equivalent vectorizability. Moreover, Akbudak and Aykanat show in [6] that the memory latency, rather than bandwidth, is the performance bottleneck for the many-core Xeon Phi system.

Intel MKL's spGEMM function also shows 1.1× to 8.9× increase in execution time with respect to that on the Xeon CPU with decreasing density. The large caches of CPUs result in significantly better sparse matrix-matrix multiplication performance of CPUs as compared to the Xeon Phi, because repeatedly accessed rows in **B** may already be available in cache. The throughput-oriented Xeon Phi architecture has much smaller caches, resulting in many inefficient reloads of data from global memory. This trend is similar to what is observed in [50] for both the CPU and the Xeon Phi.

To combat the inefficiencies of existing architectures, we design a many-core architecture using an SPMD paradigm to exploit the massive parallelism inherent in the algorithm. We allow simple, dynamic resource allocation using asynchronous tiles and a non-coherent memory system. The SPMD paradigm allows computation to drift among the cores when work is imbalanced, leading to better compute utilization than the SIMT programming model in GPUs. Furthermore, to address the performance discrepancy between the *multiply* and *merge* phases due to different memory access patterns, we employ a reconfigurable cache hierarchy *to allow data-sharing across multiple cores when it is needed and segmenting storage to isolated units when it is not.*

5. THE OUTERSPACE ARCHITECTURE

Qualitative analysis and results from the previous section reveal two key reasons why outer product multiplication does not perform well on conventional hardware:

- Outside of a particular outer product calculation during the *multiply* phase, there is no reuse of elements, as explained in Section 4. During this phase, the corresponding columns-of-**A** and rows-of-**B** can be shared across processing elements, whereas there is no data sharing between the processing units during the *merge* phase. Traditional hardware lacks the support to optimize for both of these phases, while dealing with variable memory allocation.
- While the *multiply* phase has consistent control flow between adjacent threads, dynamic execution paths in the *merge* phase necessitate fine-grained asynchrony across processing elements to fully utilize the available parallelism. An ideal architecture will allow for fully decoupled processing without sacrificing the ability to effectively share data.

To harness the massive parallelism and data reuse through fine-grained control over what data is fetched

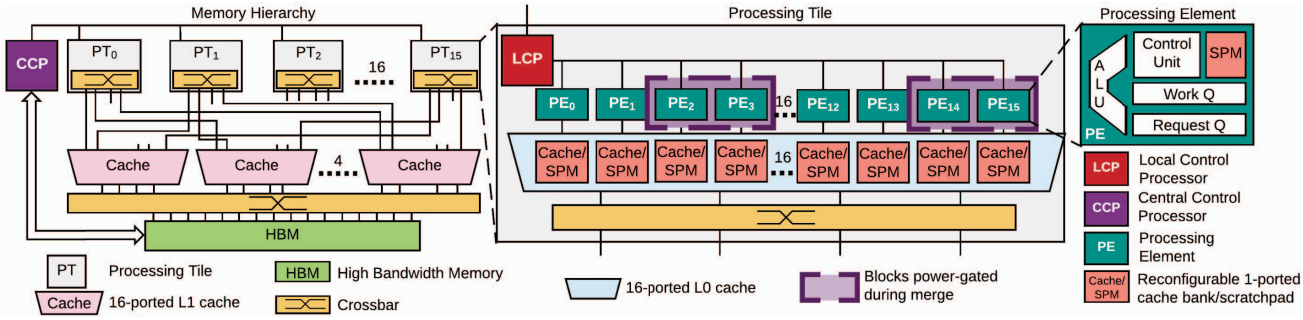

Figure 5: The memory hierarchy (left) and the architectures of the Processing Tile (center) and the Processing Element (right). The solid dark lines represent 64-bit bidirectional links.

from memory, we propose our custom architecture, OuterSPACE. Figure 5 shows the microarchitecture of the OuterSPACE system. Our architecture is a system-on-chip consisting of SPMD-style parallel processing elements arranged as tiles, with two levels of shared, reconfigurable caches, and a set of control processors for scheduling and coordination, all connected to an HBM. In this work, we implement simple cache-to-scratchpad reconfiguration by switching-off tag arrays, although recent work such as coherent scratchpads [8] and Stash [37] have the potential to make OuterSPACE more general.

Following is a summary of the key elements of our proposed architecture:

- *Processing Element (PE)*: A custom data-streaming and compute engine comprised of an ALU with a floating point unit, scratchpad memory, a control unit, a work queue and an outstanding request queue (PEs are grouped into processing tiles)
- *Local Control Processor (LCP)*: A small in-order core that coordinates the PEs within a tile, streaming instructions for the PEs to execute
- *Central Control Processor (CCP)*: A power-efficient core that is responsible for scheduling work and allocating memory for intermediate data structures
- *High-speed crossbars* and *coalescing caches* that can be reconfigured into scratchpad memories
- *High Bandwidth Memory* that stores the input matrices and the intermediate partial products

In the following subsections, we present a detailed description of each element of the OuterSPACE system. We model 16 PEs per tile and 16 tiles based on scalability studies [53] to ensure that crossbar sizes do not bottleneck our architecture.

5.1 Processing Element

A block diagram of the Processing Element (PE) is shown on the right side of Figure 5. At the core of the PE is a floating-point capable ALU for multiplication and summation of the matrix elements. These elements are streamed-in from the memory hierarchy (Section 5.3), orchestrated by the Control Unit, which generates loads and stores to memory. An outstanding request queue keeps track of the loads and stores that are in-flight. Lastly, there is a small private scratchpad and a FIFO work queue for decoded instructions and bookkeeping data, which are supplied to the PE by the LCP.

5.2 Processing Tile

A processing tile in our design consists of 16 PEs, an LCP and a reconfigurable cache with 16 processor-side read/write ports and 4 memory-side ports. These caches internally consist of 16 single-ported cache banks and a controller (not shown) that interfaces with the LCP to reconfigure the cache into a scratchpad. As mentioned in Section 4, this is the key structure that reconfigures our system from a shared memory architecture into a non-shared one. The PEs within a tile communicate with the lower memory levels through a 4-ported crossbar.

5.3 Memory Hierarchy

Figure 5 shows 16 processing tiles that interface with the main memory through 4 L1 caches. These caches act like *victim caches* [32] and thus are smaller than their L0 counterparts, in order to minimize undesired eviction of data that is going to be reused. They cache-in elements that are evicted from the L0 caches when some PEs start drifting away in their execution flow from others within a tile, which occurs when the PEs are operating on multiple rows simultaneously. The L0 caches also contain a 16×16 crossbar (not shown).

Our baseline main memory features an HBM 2.0 x64 interface [55], with 16 memory channels and a total memory bandwidth of 128 GB/s. With a PE clocked at 1.5 GHz, the total bandwidth for all the PEs is 9.2 TB/s (256 PEs × 1.5 giga-operations per second × 12 B per access for double-precision value and index pair × read + write channels). We overcome this bandwidth gap with a multi-level cache-crossbar hierarchy connecting the PEs to the memory. This hierarchy provides extensive reuse of temporally correlated data within an outer product.

The PEs in our system execute in an SPMD-fashion, often drifting apart from each other and only synchronizing at the end of the *multiply* and *merge* phases, as outlined in Section 4. This contrasts with the GPU, which traditionally employs a SIMT execution model, where compute units operate in locksteps over a dataset.

This also opens up avenues for various circuit-level techniques to improve energy efficiency, such as *voltage throttling* and *dynamic bandwidth allocation*. Furthermore, the PEs only share read-only data, which allow

for the crossbars to be non-coherent structures without breaking correctness. Incorporating techniques such as SARC [35], VIPS [36] or DeNovo [18] would help expand OuterSPACE to algorithms demanding coherence, which we defer to a future work.

5.4 Mapping the Outer Product Algorithm

This subsection illustrates how the outer product algorithm described in Section 4 maps to OuterSPACE.

5.4.1 Multiply Phase

As mentioned in Section 4.1 and illustrated in Figure 2, processing units (PEs in OuterSPACE) multiply an element of a column-of-**A** with the entire corresponding row-of-**B**. The L0 caches retain the rows-of-**B** until all the PEs within a tile are finished with this set of multiplication. The PEs store the multiplied results in contiguous memory chunks, which are part of the linked list corresponding to a row-pointer (R_i), using a write-no-allocate policy to avoid results evicting elements of **B**. The memory layout described in Figure 2, where chunks of memory in each node of the list are discontiguous, allows each PE to work independently without any synchronization. Thus, the only data that the PEs share throughout the *multiply* phase is read-only data (values and column-indices within rows-of-**B**).

5.4.2 Merge Phase

A subset of the PEs within each tile is assigned to merge all the partial products corresponding to a single row of the resultant final matrix at the start of the *merge* phase. The values to be merged are distributed across the partial products, as indicated by R_i in Figure 2.

For minimum computational complexity, a parallel merge-sort algorithm would be optimal for merging rows across partial products in $rN\log(rN)$ time, where rN is the total number of elements in the row. However, this will result in multiple re-fetches of the same data when the entire row cannot be contained within the upper memory hierarchy, which will dominate the execution time. Instead, we focus on minimizing memory traffic. Our algorithm operates as follows (assuming the number of rows to merge is rN, each of which contains rN elements, where r and N are the density and dimension of the matrix, respectively):

1. Fetch the head of each row and sort by column index into a linked list ($\mathcal{O}(r^2N^2)$ operations)
2. Store the smallest-indexed element from the list into the final location, load the next element from the corresponding row and sort it into the list ($\mathcal{O}(rN)$ operations)
3. Repeat 2 until all elements of each row have been sorted and shipped to memory (r^2N^2 iterations)

The overall complexity is $\mathcal{O}(r^3N^3)$. While less efficient algorithmically, number of elements stored in local memory is only on the order of rN. A local buffer of the next elements to sort can help hide the latency of inserting elements into the list under the latency of grabbing a new element from main memory. Given our target workloads and system specifications, the time to sort the values is expected to be on the order of one-tenth to one-millionth of the time for memory to supply the values, with sparser matrices having a smaller discrepancy.

Because this phase requires no data sharing across tiles (each set of rows that is being merged reads independent data), the shared cache can be reconfigured into private scratchpads. This minimizes memory transactions by eliminating conflict cache misses within a processing tile and saves energy by eliminating tag bank lookups. If a scratchpad bank is too small to buffer the entire merge operation of each row, we recursively merge a subset of the rows into a single row until the number of rows is sufficiently small.

Figure 5 illustrates the reconfigurability of the tiles to handle the different phases of computation. Specifically:

- A batch of the PEs within a tile are disabled to throttle bandwidth to each PE and conserve power. Half of the PEs load the row buffers, while the remainder sort the incoming values and store them to the resultant matrix.
- A subset of the cache banks within a tile are reconfigured as private scratchpad memories for the PEs (Figure 5). The reconfiguration can be achieved simply by power-gating the tag array of the cache. The address range of each bank is remapped by software, with assistance from the LCP. Thus, the reconfigured private cache-scratchpad structure maximizes memory-level parallelism (MLP), while minimizing stalls due to computation. This technique has been employed on a smaller scale in GPUs [3].

5.5 Memory Management

Precise memory pre-allocation for the intermediate partial products is impossible, as the sizes of the outer products are dependent on the specific row/column sizes [41]. However, due to the predictable nature of the two phases, we can greatly reduce the overhead of dynamic memory allocation over general schemes.

For the *multiply* phase, we statically assign each partial product enough storage to handle the average case (the average number of non-zero elements can be quickly calculated from the compressed format before computation begins), as well as a large *spillover stack* to be used dynamically for larger products. As a statically assigned PE (one per row/column pair) begins computation of a given product, it can evaluate from the row-pointers exactly how much spillover space is needed. The PE sends a single atomic instruction to increment a global stack pointer by the appropriate amount, and writes the current value location visible to the other PEs. As long as the PEs do not consume the entirety of their static allocation before the atomic load returns, the latency of memory allocation can be entirely hidden.

In the *merge* phase, we perform a single memory allocation before computation by maintaining a set of counters for the size of each row in the multiply phase. Data is then streamed from two contiguous memory segments for each partial product row: the static partition from the *multiply* phase and, if used, a portion of the

Table 2: Simulation parameters of OuterSPACE.

Processing Element	1.5 GHz clock speed, 64-entry outstanding requests queue, 1 kB scratchpad memory *Multiply* phase: All 16 PEs per tile active *Merge* phase: 8 PEs per tile active, rest disabled
L0 cache/ scratchpad	*Multiply* phase: 16 kB, 4-way set-associative, 16-ported, shared, non-coherent cache with 32 MSHRs and 64 B block size per tile *Merge* phase: 2 kB, 4-way set-associative, single-ported, private cache with 8 MSHRs and 64 B block size + 2 kB scratchpad per active PE-pair
L1 cache	4 kB, 2-way set-associative, 16-ported, shared, non-coherent with 32 MSHRs and 64 B blocks
Crossbar	16×16 & 4×4 non-coherent, swizzle-switch based
Main Memory	HBM 2.0 with 16 64-bit pseudo-channels each @ 8000 MB/s with 80-150 ns average access latency

spillover space. The data is merged and streamed-out to the separately-allocated *merge* phase storage. As matrices get sparser, the amount of space wasted due to merge collisions in this space becomes negligibly small. We discuss further about allocation overheads in Section 7.3.

The memory footprint of the outer product approach can be represented as $(\alpha \cdot N + \beta \cdot N^2 \cdot r + \gamma \cdot N^3 \cdot r^2)$, where α, β and γ are small, implementation-dependent constants, for uniformly random sparse matrices with dimension N and density r. For non-uniform matrices, this metric is not easily quantifiable, as the sparsity patterns of the two matrices heavily influence the number of collisions between non-zeros.

5.6 Other Matrix Operations

We evaluate a sparse-matrix sparse-vector multiplication algorithm similar to our matrix-matrix implementation, with a few simplifications. In particular, the amount of work assigned to each PE is reduced and no scratchpad is needed in the *merge* phase, as partial products do not need to be sorted.

Element-wise matrix operations follow a similar procedure as the *merge* phase of the matrix-matrix multiplication algorithm described in Section 4.2. Given N matrices $\mathbf{A}_1$, $\mathbf{A}_2$, ..., $\mathbf{A}_N$ with the same dimensions, the data can be reorganized into a data structure similar to the one illustrated in Figure 2 and element-wise operations (+, -, ×, /, ==) can be performed on it.

There is close to a one-to-one correspondence between data operations in each of the typical element-wise matrix routines (addition, subtraction, multiplication and comparison) and the *merge* phase of outer product sparse matrix-matrix multiplication. Thus, they are expected to have similar complexities and we do not evaluate them separately in this work.

6. EXPERIMENTAL SETUP

To evaluate the performance of the outer product algorithm on OuterSPACE, we modeled the key elements, namely, the PEs, the cache-crossbar hierarchy and the HBM, using the gem5 simulator [13]. We created two separate models pertaining to the two phases. We ignored the start-up time, since it can be easily hidden by the latency of a few memory accesses, and scheduling delays. We also assumed that the PEs are greedily scheduled for the *multiply* phase. We modeled an outstanding request queue with 64 entries, for each PE, which is at par with the number of load-store units in modern GPUs [1] and CPUs [29], in order to hide latencies of memory accesses. The simulation parameters used are shown in Table 2.

Table 3: CPU and GPU configurations.

CPU	3.6 GHz Intel Xeon E5-1650V4, 6 cores/12 threads 128 GB RAM, solid state drives
GPU	NVIDIA Tesla K40, 2880 CUDA cores @ 745 MHz, 12 GB GDDR5 at 288 GB/s

We chose our parameters to optimize for performance and power efficiency. For the *merge* phase, we enabled only 8 of the 16 PEs per tile and reconfigure a proportional number of cache banks into private scratchpad memories. We, in fact, observed that enabling a greater number of PEs results in slight performance degradation due to thrashing in the L1 cache. The scratchpad size was chosen to be large enough to hide the load latency during sort operations.

CACTI 6.5 [45] was used for modeling cache latency, area and power values. For power dissipated by the core, we used static and dynamic power consumption values for an ARM Cortex-A5 with VFPv4 in 32 nm from [53]. We pessimistically used the same aggressive core model for the PEs in addition to the LCPs and CCP. Dynamic power consumption was calculated by capturing activity factors of the cache and cores from simulation. The HBM power was derived from the JEDEC specification document [55] and [7]. The parameters for modeling the crossbars were obtained from [53].

We built an instruction trace generator for the PEs and ran the generated traces through our gem5 model in order to process large matrices. Due to the practical limitations of this approach, we did not model the dynamic memory allocation overhead in OuterSPACE, and thus do not consider this overhead across any other platform in our evaluation. However, in Section 7.3, we provide an analysis of this overhead by quantifying the number of dynamic allocation requests using the allocation approach in Section 5.5.

7. EVALUATION

We evaluate the OuterSPACE architecture by comparing against state-of-the-art library packages on commercial systems, namely, Intel MKL (Version 2017 Initial Release) on the CPU, NVIDIA cuSPARSE (Version 8.0) and CUSP (Version 0.5.1) on the GPU. The specifications of these hardware are summarized in Table 3. We show the performance of OuterSPACE on two important classes of matrix operations, sparse matrix-matrix and sparse matrix-vector multiplication.

We report the simulation times obtained from our gem5 models for the *multiply* and *merge* models running instruction traces that exclude memory-allocation code.

In order to provide fair basis for comparison against the CPU and the GPU, we discard memory allocation time and only consider the execution time of computation functions for the MKL, cuSPARSE and CUSP implementations (memory-allocation and computation are

Table 4: Matrices from University of Florida SuiteSparse (SS) [21] and Stanford Network Analysis Project (SNAP) [39] with their plots, dimensions, number of non-zeros (nnz), average number of non-zeros per row/column (nnz_{av}) and problem domain.

Matrix	Plot	Dim. nnz nnz_{av}	Kind
2cubes_-sphere		101K 1.6M 16.2	EM problem
amazon-0312		401K 3.2M 8.0	Co-purchase network
ca-Cond-Mat		23K 187K 8.1	Conden-sed matter
cage12		130K 2.0M 15.6	Directed weighted graph
cit-Patents		3.8M 16.5M 4.4	Patent citation network
cop20k-_A		121K 2.6M 21.7	Accele-rator design
email-Enron		36.7K 368K 10.0	Enron email network
facebook		4K 176K 43.7	Friend-ship network
filter3D		106K 2.7M 25.4	Reduc-tion problem
m133-b3		200K 801K 4.0	Combi-natorial problem
mario-002		390K 2.1M 5.4	2D/3D problem
offshore		260K 4.2M 16.3	EM problem
p2p-Gnutella-31		63K 148K 2.4	p2p network
patents_-main		241K 561K 2.3	Directed weighted graph
poisson-3Da		14K 353K 26.1	Fluid Dynam-ics
roadNet-CA		2.0M 5.5M 2.8	Road network
scircuit		171K 959K 5.6	Circuit simula-tion
webbase-1M		1M 3.1M 3.1	Directed weighted graph
web-Google		916K 5.1M 5.6	Google web graph
wiki-Vote		8.3K 104K 12.5	Wiki-pedia network

discrete functions for these libraries). While reporting throughput, we only consider operations associated with multiplication and accumulation in order to maintain consistency across algorithms and to avoid artificially inflating performance by accounting for additional book-keeping. We verify that the raw GFLOPS reported by our GPU performance counters on the Florida benchmark suite (Section 7.1.2) are similar to those reported by Liu and Vinter [41], but omit the results for brevity.

7.1 Sparse Matrix-Matrix Multiplication

Without loss of generality, we evaluate the performance of sparse matrix-matrix multiplication on our platform by multiplying a sparse matrix with itself ($\mathbf{C} = \mathbf{A}\times\mathbf{A}$; $\mathbf{A}$ in CR format to begin with, generating $\mathbf{C}$ in CR), in order to closely mimic the multiplication of two matrices of similar sizes/densities. We use two different sets of matrices, *synthetic* and *real-world*, as benchmarks to evaluate OuterSPACE. We account for format conversion overheads for non-symmetric matrices (Section 4.3) while reporting performance results for OuterSPACE, in order to model the worst-case scenario.

7.1.1 Synthetic Matrices

In Figure 6, we compare the performance-scaling of the outer product algorithm on OuterSPACE against the CPU and GPU libraries, using a set of synthetic datasets obtained from the Graph500 R-MAT data generator [46]. The R-MAT parameters were set to their default values (A=0.57, B=C=0.19) used for Graph500 to generate undirected power-law graphs, which is also employed in recent work in graph analytics [51] [58]. We present results for medium-sized matrices corresponding to `nEdges` equal to 100,000 with `nVertices` swept between 5,000 and 80,000. In order to illustrate the impact of sparsity pattern on performance, we also provide comparisons against uniformly random matrices of same dimensions and densities.

OuterSPACE performs consistently well with respect to other platforms. It outperforms MKL and CUSP with a greater margin for the R-MATs than for the uniformly random matrices, with the execution time changing only slightly across matrix densities. cuSPARSE, on the other hand, performs better with increasing density. OuterSPACE exhibits slight performance degradation with increasing density for uniformly random matrices, but gets starker speedups over the GPU for power-law graphs. This data also substantiates that the outer product algorithm is much less sensitive to a change in size for a given number of non-zeros than the MKL, which correlates with our observations on the CPU (Figure 3).

7.1.2 Real-World Matrices

The real-world matrices we evaluate are derived from the University of Florida SuiteSparse Matrix Collection [21] and the Stanford Network Analysis Project (SNAP) [39], containing a wide spectrum of real-world sparse matrices from diverse domains such as structural engineering, computational fluid dynamics, model reduction, social networks, web graphs, citation networks, etc. These matrices have been widely used for performance evaluation in prior work in this area [20, 26, 42, 52]. We choose matrices from this collection that have both *regular* and *irregular* structures. Table 4 presents a summary of the structure and properties of these matrices, which have dimensions varying from 4,096 to 3,774,768 and densities ranging between 0.001% and 1.082%.

In Figure 7, we present the speedups of OuterSPACE over the MKL, cuSPARSE and CUSP. OuterSPACE steadily achieves speedups across all the matrices identi-

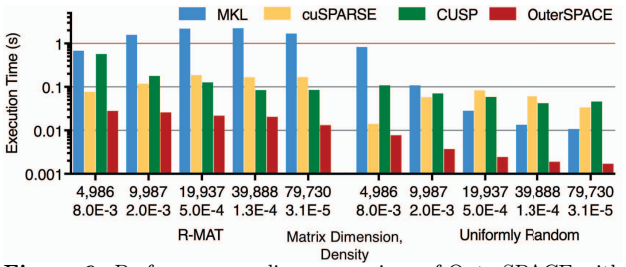

Figure 6: Performance-scaling comparison of OuterSPACE with change in matrix dimension and density. The set of data on the left is for R-MATs with parameters (A=0.57, B=C=0.19) for undirected graphs. The set on the right is for uniformly random matrices of the same size and density as the R-MATs.

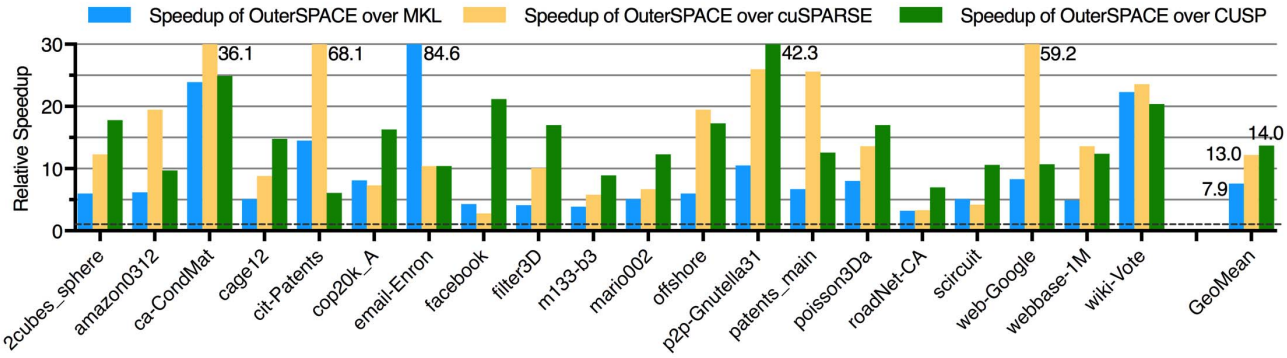

Figure 7: Speedups of OuterSPACE over the CPU running Intel MKL and the GPU running cuSPARSE and CUSP.

fied in Table 4, with an average of 7.9× over the Intel MKL, 13.0× over cuSPARSE and 14.0× over CUSP.

Some of the matrices that show relatively lesser speedups over the MKL and cuSPARSE are *filter3D* and *roadNet-CA*. These matrices are regular (*i.e.* have most of their non-zeros along their diagonals) and work better with the multiplication algorithms used by the libraries, because they incur fewer comparisons while multiplying two regular matrices. OuterSPACE performs only 3.9× better than MKL for the *m133-b3* matrix due to the uneven distribution of non-zeros along the columns, which leads to load imbalances during the *merge* phase and uneven data sharing patterns during the multiply phase. MKL performs particularly bad on *email-Enron*, a real-world email dataset with the characteristics of a power-law graph [17], substantiating the observation made in Section 7.1.1. OuterSPACE also achieves the highest speedups over cuSPARSE for matrices that have a more smeared (irregular) distribution of non-zeros, such as *ca-CondMat*, *cit-Patents*, *p2p-Gnutella31* and *web-Google* (Table 4).

OuterSPACE running the outer product algorithm over this suite of matrices achieves an average throughput of 2.9 GFLOPS, accounting only for useful operations (multiplications and summations). We also observe a memory bandwidth utilization of 59.5-68.9% for the multiply phase and a slightly lower 46.5-64.8% for the merge phase. This is due to fewer active PEs and greater use of local memory during the merge phase. This can be further improved if the matrices are strategically laid out in memory such that there is fairer access to every memory channel, but this would require compiler support and is beyond the scope of our work.

7.2 Sparse Matrix-Vector Multiplication

In this section, we evaluate the performance of sparse matrix-vector multiplication on OuterSPACE. Table 5 shows the speedups of OuterSPACE over the CPU running MKL and GPU running cuSPARSE with vector densities ranging from 0.01 to 1.0 (fully dense). The performance of MKL is constant across different vector densities for a given matrix dimension and density, because MKL's sparse matrix-vector method performs the best when the vector is treated as a dense vector regardless of the number of zeros in the vector. cuSPARSE, however, scales with change in vector density.

Table 5: Speedups of OuterSPACE over CPU (MKL) and GPU (cuSPARSE) for sparse matrix-vector multiplication. The density of the vector (r) is varied from 0.01 to 1.0. The sparse matrices contain uniformly random distribution of one million non-zeros.

	Speedup over CPU			Speedup over GPU		
Matrix Dim.	**r= 0.01**	**r= 0.1**	**r= 1.0**	**r= 0.01**	**r= 0.1**	**r= 1.0**
65,536	93.2	8.7	0.8	92.5	11.2	3.8
131,072	107.5	9.9	1.0	98.2	11.2	2.8
262,144	152.1	12.6	1.2	126.0	12.5	2.3
524,287	196.3	17.2	1.7	154.4	17.4	2.2

Across all matrix sizes, the speedup of OuterSPACE scales linearly with vector density, with a 10× reduction in density resulting in approximately a 10× gain in speedup. Such performance scaling is possible because the outer product algorithm only accesses specific columns in the input matrix that match the indices of the non-zero elements of the vector. This eliminates all redundant accesses to the matrix that are present in a conventional inner product algorithm. Thus, the number of memory accesses to the sparse matrix is directly proportional to the number of non-zeros in the vector.

We identify two striking differences between the outer product algorithm on OuterSPACE and existing matrix-vector multiplication on CPUs and GPUs. First, unlike conventional algorithms where the performance depends heavily on the dimensions of the matrix regardless of the density, the performance of the outer product algorithm scales with the number of non-zero elements in the matrix, while remaining independent of the matrix dimension, for uniformly random matrices. Second, the performance of the sparse matrix-vector algorithm on OuterSPACE also scales linearly with the density of the vector, which allows OuterSPACE to outperform traditional algorithms for sparse vectors. Even while working on small, dense vectors, OuterSPACE achieves within 80% of the MKL's performance, as reflected in the fourth column of Table 5.

7.3 Dynamic Memory Allocation

As detailed in Section 5.5, the latency of dynamic allocation requests by the PEs can typically be hidden by sending the atomic request to increment the spill-over pointer before the PE begins multiplication. Increasing the amount of space statically assigned for

partial products lowers the execution time by decreasing the number of accesses for dynamic allocation, at the expense of wasted storage, illustrating a common performance-storage trade-off.

We assume $\alpha \cdot \frac{nnz^2}{N}$ elements are allocated statically, where $\frac{nnz^2}{N}$ is the amount of storage needed for and average row and α is a parameter. Our analysis of the total number of dynamic requests to increment the spill-over pointer, while sweeping (α), shows that the count of these requests drops to less than 10,000 for α>=2 for almost all the matrices in Table 4. *m133-b3* is an outlier, with zero dynamic requests, as it has exactly 4 non-zeros per row, which fits within the statically allocated space even for α=1. Our strategy works best for matrices that are more uniformly distributed, since a suitable value of α can eliminate most of the dynamic allocation requests. However, this overhead for real-world matrices is largely dependent on the sparsity patterns of the matrices.

7.4 Power and Area Analysis

Table 6 presents the area and power estimates for OuterSPACE in 32 nm. The total chip area, excluding the HBM controllers, is calculated to be 87 mm^2. The power consumption of the system is calculated to be 24 W, using the parameters presented in Section 6. This yields on average 0.12 GFLOPS/W for OuterSPACE.

Table 6: Power and area estimates for OuterSPACE (Figure 5).

Component	Area (mm^2)	Power (W)
All PEs, LCPs, CCP	49.14	7.98
All L0 caches/scratchpads	34.40	0.82
All L1 caches	3.13	0.06
All crossbars	0.07	0.53
Main memory	N/A	14.60
Total	**86.74**	**23.99**

For comparison, the mean measured power consumption of the K40 GPU while running the workloads was 85 W. With the GPU achieving only 0.067 GFLOPS on an average, this yields 0.8 MFLOPS/W. The OuterSPACE system is, thus, approximately 150× better than the GPU on the performance/power metric.

8. OUTERSPACE SCALING

Our current architecture is still well below current reticle sizes and power limitations. In order to handle matrix sizes larger than a few million, a silicon-interposed system with 4 HBMs and 4× the PEs on-chip could be realized. This extended configuration would support matrices containing tens to hundreds of millions of non-zero elements, limited by the capacity of the HBM. In order to process larger matrices, we conceive equipping our architecture with node-to-node serializer-deserializer (SerDes) channels to allow multiple OuterSPACE nodes connected in a torus topology, thus minimizing system latency and maximizing throughput. Such a system would be able to process matrices with billions of non-zeros. To scale to problems involving matrices with trillions of non-zeros, we envision interconnecting many such 16 OuterSPACE-node clusters.

9. RELATED WORK

With the prevalence of sparse matrix operation kernels in big data applications and their rising significance in a multitude of areas, there has been abundant work on accelerating sparse matrix-dense vector/matrix multiplication [5, 12, 28, 43]. However, there has been relatively less work done to accelerate sparse matrix-sparse vector/matrix multiplication and more on creating software frameworks for existing state-of-the-art architectures like multi-core and many-core CPUs [6, 52, 57], GPUs [20, 26, 41, 43] and heterogeneous (CPU-GPU) systems [41, 42, 43]. On the contrary, our work demonstrates an efficient co-design of outer product based sparse matrix multiplication with our custom, scalable, reconfigurable architecture, achieving significant speedups over state-of-the-art CPU and GPU libraries.

There has been some work on enhancing the underlying hardware for sparse matrix-matrix multiplication. Lin *et al.* [40] propose an FPGA-based architecture for sparse matrix-matrix multiplication that uses on-chip dedicated DSP blocks and reconfigurable logic as processing elements (PEs). However, the design is significantly limited by scarce on-chip FPGA resources, including the number of PEs and the size of the on-chip memory. Yavits and Ginosar [64] explore a juxtaposed Resistive CAM and RAM based sparse matrix-matrix and sparse matrix-vector accelerator, applying a row-by-row algorithm to efficiently match the indices of the multiplier and multiplicand and select the ReRAM row, where the corresponding non-zero element of the sparse multiplicand matrix/vector is stored. Zhu *et al.* [66] introduce a 3D-stacked logic-in-memory system by placing logic layers between DRAM dies to accelerate a 3D-DRAM system for sparse data access and build a custom CAM architecture to speed-up the index-alignment process of column-by-column matrix multiplication by taking advantage of its parallel matching characteristics.

However, the aforementioned hardware solutions accelerate SpGEMM algorithms with large amount of redundant memory accesses, which we have identified to be a key performance bottleneck in sparse matrix-matrix multiplication. With our custom architecture, OuterSPACE, tailored for the outer product algorithm which eliminates most of these redundant accesses, we achieve significant speedups as well as high energy efficiency (Section 7).

There also exists work for accelerating sparse matrix-vector multiplication. Mishra *et al.* [44] add blocking to the baseline software and design fine-grained accelerators that augment each core in sparse matrix-vector multiplication. Nurvitadhi *et al.* [47] propose a SpM-SpV algorithm using column-based SpMV, row blocking, column skipping, unique value compression (UVC), and bit-level packing of matrix data and a hardware accelerator for it, composed of a data management unit and PEs. In our work, we address both sparse matrix-matrix multiplication and sparse matrix-vector multiplication.

10. CONCLUSION

The Intel MKL and NVIDIA CUSP and cuSPARSE libraries are successful state-of-the-art libraries for sparse

linear algebra. However, our experiments and analyses show that MKL and cuSPARSE work optimally only for regular sparse matrices. While CUSP is insensitive to the irregularity of sparse matrices, it introduces extra memory overheads for the intermediate storage [41].

In this work, we explore an outer product based matrix multiplication approach and implement it on traditional CPUs and GPUs. We discover inefficiencies in these architectures, which lead to sub-optimal performance of the outer product algorithm, and resolve them by building a new custom reconfigurable hardware. Our novelty lies in the *efficient co-design of the algorithm implementation and the hardware*. We demonstrate OuterSPACE's efficiency for two key kernels, sparse matrix-matrix multiplication and sparse matrix-vector multiplication, which form the building blocks of many major applications.

The reconfigurable memory hierarchy of OuterSPACE, which adapts to the contrary data-sharing patterns of the outer product algorithm, aids in reducing the number of main memory accesses. This, coupled with the increased flexibility across PEs through SPMD-style processing (due to lack of synchronization and coherence overheads), enables OuterSPACE to achieve good throughput and high speedups over traditional hardware. Energy savings are attributed to the bare-bone PE and energy-efficient swizzle-switch crossbar designs [53].

In essence, our work demonstrates that a massively-parallel architecture consisting of asynchronous worker cores, coupled with memory hierarchies that are tailored to retain reusable data, uncovers an enormous potential to accelerate kernels that are heavily memory-bound.

REFERENCES

[1] "NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Kepler^{TM} GK110," NVIDIA, Tech. Rep., 2012. [Online]. Available: https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[2] (2014) cuSPARSE Library. https://developer.nvidia.com/cuSPARSE.

[3] "NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Fermi^{TM}," http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, NVIDIA, Tech. Rep., 2015.

[4] "Seventh Green Graph 500 List," 2016. [Online]. Available: http://green.graph500.org/lists.php

[5] S. Acer, O. Selvitopi, and C. Aykanat, "Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems," *Parallel Computing*, vol. 59, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819116301041

[6] K. Akbudak and C. Aykanat. (2017) Exploiting Locality in Sparse Matrix-Matrix Multiplication on Many-Core Architectures. http://www.prace-ri.eu/IMG/pdf/wp144.pdf.

[7] M. Alfano, B. Black, J. Rearick, J. Siegel, M. Su, and J. Din, "Unleashing Fury: A New Paradigm for 3-D Design and Test," *IEEE Design & Test*, vol. 34, no. 1, pp. 8–15, 2017.

[8] L. Alvarez, L. Vilanova, M. Moreto, M. Casas, M. Gonzàlez, X. Martorell, N. Navarro, E. Ayguadé, and M. Valero, "Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures," in *Proceedings of the 42nd Annual Int'l Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 720–732.

[9] A. Azad, A. Buluç, and J. R. Gilbert, "Parallel Triangle Counting and Enumeration Using Matrix Algebra," *2015 IEEE Int'l Parallel and Distributed Processing Symposium Workshop*, pp. 804–811, 2015.

[10] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM J. Scientific Comput.*, 2011.

[11] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[12] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari, "Optimal Sparse Matrix Dense Vector Multiplication in the I/O-Model," *Theory of Computing Systems*, vol. 47, no. 4, pp. 934–962, Nov 2010.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[14] S. Brin and P. L, "The anatomy of a large-scale hypertextual web search engine," *7th Int'l WWW Conference*, 1998.

[15] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design implementation and applications," *The Int'l Journal of High Performance Computing Applications*.

[16] A. Buluç and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *IEEE Int'l Symposium on Parallel and Distributed Processing*, ser. IPDPS '08. IEEE, 2008, pp. 1–11.

[17] A. Chapanond, M. S. Krishnamoorthy, and B. Yener, "Graph Theoretic and Spectral Analysis of Enron Email Data," *Computational & Mathematical Organization Theory*, vol. 11, no. 3, pp. 265–281, Oct 2005.

[18] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *2011 Int'l Conference on Parallel Architectures and Compilation Techniques*, Oct 2011, pp. 155–166.

[19] S. Dalton, N. Bell, L. Olson, and M. Garland. (2015) Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. Version 0.5.1.

[20] S. Dalton, L. Olson, and N. Bell, "Optimizing Sparse Matrix-Matrix Multiplication for the GPU," *ACM Trans. Math. Softw.*, vol. 41, no. 4, pp. 25:1–25:20, Oct. 2015.

[21] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[22] I. S. Duff, M. A. Heroux, and R. Pozo, "An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum," *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, no. 2, pp. 239–267, 2002.

[23] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "A Unified Framework for Numerical and Combinatorial Computing," *Computing in Science & Engineering*, vol. 10, no. 2, pp. 20–25.

[24] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "High-performance Graph Algorithms from Parallel Sparse Matrices," *Proc. of the Int'l Workshop on Applied Parallel Computing*, 2006.

[25] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the performance of sparse matrix-vector multiplication," in *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, ser. PDP '08. IEEE, 2008.

[26] F. Gremse, A. Höfter, L. O. Schwen, F. Kiessling, and U. Naumann, "GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. C54–C71, 2015.

[27] V. Hapla, D. Horák, and M. Merta, *Use of Direct Solvers in TFETI Massively Parallel Implementation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 192–205.

[28] A. F. Heinecke, "Cache Optimised Data Structures and Algorithms for Sparse Matrices," B.S. thesis, Technical University of Munich, April 2008.

[29] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, June 2011, no. 248966-025.

[30] S. Itoh, P. Ordejón, and R. M. Martin, "Order-N tight-binding molecular dynamics on parallel computers," *Computer Physics Communications*, vol. 88, no. 2, pp. 173–185, 1995.

[31] R. W. Johnson, C. H. Huang, and J. R. Johnson, "Multilinear algebra and parallel programming," *The Journal of Supercomputing*, vol. 5, no. 2, pp. 189–217, 1991.

[32] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual Int'l Symposium on Computer Architecture*, ser. ISCA '90. IEEE, 1990.

[33] H. Kaplan, M. Sharir, and E. Verbin, "Colored intersection searching via sparse rectangular matrix multiplication," *SCG '06: Proceedings of the twenty-second annual symposium on computational geometry*, pp. 52–60, 2006.

[34] G. Karypis, A. Gupta, and V. Kumar, "A Parallel Formulation of Interior Point Algorithms," in *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 204–213.

[35] S. Kaxiras and G. Keramidas, "SARC Coherence: Scaling Directory Cache Coherence in Performance and Power," *IEEE Micro*, vol. 30, no. 5, pp. 54–65, Sept 2010.

[36] S. Kaxiras and A. Ros, "Efficient, snoopless, System-on-Chip coherence," in *2012 IEEE Int'l SOC Conference*, Sept 2012, pp. 230–235.

[37] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have your scratchpad and cache it too," in *Proceedings of the 42nd Annual Int'l Symposium on Computer Architecture*, ser. ISCA '15, June 2015, pp. 707–719.

[38] E. Lee, H. Kang, H. Bahn, and K. G. Shin, "Eliminating Periodic Flush Overhead of File I/O with Non-Volatile Buffer Cache," *IEEE Transactions on Computers*, vol. 65, no. 4, pp. 1145–1157, April 2016.

[39] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[40] C. Y. Lin, Z. Zhang, N. Wong, and H. K. H. So, "Design space exploration for sparse matrix-matrix multiplication on FPGAs," in *2010 Int'l Conference on Field-Programmable Technology*, Dec 2010, pp. 369–372.

[41] W. Liu and B. Vinter, "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data," in *2014 IEEE 28th Int'l Parallel and Distributed Processing Symposium*, May 2014, pp. 370–381.

[42] W. Liu and B. Vinter, "A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors," *CoRR*, vol. abs/1504.05022, 2015.

[43] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli, "Sparse matrix-matrix multiplication on modern architectures," in *2012 19th Int'l Conference on High Performance Computing*, Dec 2012, pp. 1–10.

[44] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr, "Fine-grained accelerators for sparse machine learning workloads," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2017, pp. 635–640.

[45] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches."

[46] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," 2010.

[47] E. Nurvitadhi, A. Mishra, and D. Marr, "A sparse matrix vector multiply accelerator for support vector machine," in *2015 Int'l Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Oct 2015, pp. 109–116.

[48] G. Penn, "Efficient transitive closure of sparse matrices over closed semirings," *Theoretical Computer Science*, vol. 354, no. 1, pp. 72–81, 2006.

[49] M. O. Rabin and V. V. Vazirani, "Maximum matchings in general graphs through randomization," *Journal of Algorithms*, vol. 10, no. 4, pp. 557–567, 1989.

[50] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jüngel, and S. Selberherr, "ViennaCL-Linear Algebra Library for Multi- and Many-Core Architectures," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S412–S439, 2016.

[51] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the 2014 ACM SIGMOD Int'l conference on Management of data*. ACM, 2014, pp. 979–990.

[52] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi," *CoRR*, vol. abs/1302.1078, 2013.

[53] K. Sewell, R. G. Dreslinski, T. Manville, S. Satpathy, N. Pinckney, G. Blake, M. Cieslak, R. Das, T. F. Wenisch, D. Sylvester, D. Blaauw, and T. Mudge, "Swizzle-Switch Networks for Many-Core Systems," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 2, no. 2, pp. 278–294, June 2012.

[54] V. B. Shah, "An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity," Ph.D. dissertation, June 2007.

[55] A. Shilov. (2016) JEDEC Publishes HBM2 Specification. http://www.anandtech.com/show/9969/jedec-publishes-hbm2-specification.

[56] A. Smith, "6 new facts about Facebook," Feb 2014. [Online]. Available: http://www.pewresearch.org/fact-tank/2014/02/03/6-new-facts-about-facebook/

[57] P. D. Sulatycke and K. Ghose, "Caching-efficient multithreaded fast multiplication of sparse matrices," in *Proceedings of the First Merged Int'l Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, Mar 1998, pp. 117–123.

[58] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[59] A. Tech. (2013) NVIDIA Launches Tesla K40. http://www.anandtech.com/show/7521/nvidia-launches-tesla-k40.

[60] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm."

[61] S. van Dongen, "Graph Clustering by Flow Simulation," Ph.D. dissertation, 2000.

[62] Vuduc, Richard W and Moon, Hyun Jin, "Fast sparse matrix-vector multiplication by exploiting variable block structure." Springer, 2005.

[63] I. Yamazaki and X. S. Li, "On techniques to improve robustness and scalability of a parallel hybrid linear solver," *Proceedings of the 9th Int'l meeting on high performance computing for computational science*, pp. 421–434, 2010.

[64] L. Yavits and R. Ginosar, "Sparse Matrix Multiplication on CAM Based Accelerator," *CoRR*, vol. abs/1705.09937, 2017. [Online]. Available: http://arxiv.org/abs/1705.09937

[65] R. Yuster and U. Zwick, "Detecting short directed cycles using rectangular matrix multiplication and dynamic programming," *Proceedings of the 15th annual ACM-SIAM symposium on discrete algorithms*, 2004.

[66] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware," in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2013, pp. 1–6.